A Machine Learning Approach to Mapping Streaming Workloads to Dynamic Multicore Processors

A Machine Learning Approach to Mapping Streaming Workloads to Dynamic Multicore Processors

Paul-Jules Micolet
University of Edinburgh
p.r.r.micolet@ed.ac.uk

Aaron Smith
Microsoft Research
University of Edinburgh
aaron.smith@microsoft

Christophe Dubach
University of Edinburgh
christophe.dubach@ed.ac.uk

Abstract

Dataflow programming languages facilitate the design of data intensive programs such as streaming applications commonly found in embedded systems. They also expose parallelism that can be exploited using multicore processors which are now part of the mobile landscape. In recent years a shift has occurred towards heterogeneity (e. g. ARM big.LITTLE) and reconfigurability. Dynamic Multicore Processors (DMPs) bridge the gap between fully reconfigurable processors and homogeneous multicore systems. They can re-allocate their resources at runtime to create larger more powerful logical processors fine-tuned to the workload.

Unfortunately, there exists no accurate method to determine how to partition the cores in a DMP among application threads. Often programmers rely on analyzing the application manually and using a set of hand picked heuristics. This leads to sub-optimal performance, reducing the potential of DMPs. What is needed is a way to determine the optimal partitioning and grouping of resources to maximize performance.

As a first step, this paper studies the effect of thread partitioning and hardware resource allocation on a set of StreamIt applications. We show that the resulting space is not trivial and exhibits a large performance variation depending on the combination of parameters. We introduce a machine-learning based methodology to tackle the space complexity. Our machine-learning model is able to directly predict the best combination of parameters using static code features. The predicted set of parameters leads to performance on-par with the best performance found in a space of more than 32,000 configurations per application.

Categories and Subject Descriptors C.1.3 [***Computer Systems Organization***]: Other Architecture Styles— Heterogeneous, Data-Flow Architectures; D.3.4 [***Programming Languages***]: Processors—Code Generation, Optimization

Keywords Machine Learning, Dynamic Multicore Processor, Streaming Programming Languages

1. Introduction

Multicore processors are now common in all computing systems ranging from mobile devices to data centers. As advances in single threaded performance have slowed, multicore processors have offered a way to use the increasing numbers of transistors available. However, designing processors that scale to a large number of cores is difficult and a shift towards tiled architecture seems inevitable. A tiled architecture such as Tilera [2] or Raw [22] is composed of smaller simpler cores that are placed on a regular grid. This improves hardware scalability and enables multi-threaded applications to exploit the large core count.

However, workloads that require high single threaded performance are penalized by the simple nature of each core [7]. One solution to this problem is heterogeneous multicores which utilize cores with different levels of power and performance. Although heterogeneous multicores are common place in mobile devices, they have little reconfiguration or adaptive capabilities (e. g. only two type of cores available for ARM big.LITTLE). Dynamic multicore processors offer a solution to this problem by allowing cores to compose (or fuse) together [12] into larger logical cores to accelerate single threads. This produces "on-demand" heterogeneity where cores are grouped to adapt to the workload's demand.

While dynamic multicore processors sound like a promising approach, they come with their own challenges, particularly on the software side [25]. In most parallel programming models such as OpenMP, the user is directly responsible for mapping parallelism to the hardware; a difficult and time consuming task. This problem is further exacerbated when hardware resources can be combined since programmers have to take into account the dynamic behavior of the architecture [3].

To solve this problem, we first argue that there is a need to raise the programming abstraction and remove the burden of mapping parallelism from programmers. Dataflow programming models such as StreamIt [20] and Lime [1] offer one part of the solution. Applications are expressed as dataflow graphs and — ideally — the compiler or runtime determines the mapping of parallelism onto the available hardware and controls the grouping of hardware resources. However, optimally mapping parallelism and managing hardware resources remains an open problem given the sheer complexity of the resulting design space.

In this paper, we first conduct an analysis of the design space and show the impact of modifying resources and thread mapping. We conduct this analysis using a set of StreamIt programs and run them on a verified cycle-level simulator for a tiled reconfigurable architecture with support for core composition. We develop a machine learning model using the information gathered from our exploration. This model predicts the best number of threads for a given application and an optimal number of cores to allocate to each thread.

To demonstrate the viability of our approach we compare the results of the predictive model to the best sampled thread and core composition pairing in a space of more than 32,000 design points. The model matches, and even outperforms in some cases, the performance of the best sampled points in the space, with speedups of up to 9x on a 16 core processor compared to single threaded execution on a single core.

The main contributions of this paper are:

- An analysis of the co-design space of thread partitioning and core composition;
- A study on the impact of a simple loop transformation on the optimal core composition;
- A machine-learning model to determine the optimal core composition and thread partitioning;
- An analysis of the most important static code features used by the model.

The rest of the paper is structured as follow. Section 2 presents information on dynamic multicore processors and dataflow programming models. Section 3 motivates this work by showing the complexity of the design space. Section 4 describes our methodology and section 5 presents an in-depth analysis of the design space. Section 6 develops a machine-learning model to predict the best thread mapping and core composition while Section 7 shows the performance achieved by our model. Related work is discussed in Section 8 and Section 9 concludes this paper.

2. Background

This section reviews the main features of a dynamic multicore processor. It also briefly introduces streaming programming models and their relevance to dynamic multicore processors.

2.1 Dynamic Multicore Processors

Chip Multiprocessors (CMPs) have become ubiquitous due to the difficulty in scaling single core performance. CMPs with homogeneous cores have dominated the space as they reduce the complexity of the design problem. Yet research shows that using heterogeneous cores allows for better performance [18], albeit with increased design complexity. In both cases, once the chip is fabricated, the design cannot be modified, meaning that many of the trade-offs between power, performance and area cannot be changed later on.

Dynamic Multicore Processors (DMPs) attempt to bridge the gap between the two previous designs by allowing the execution substrate to adapt dynamically at runtime. A DMP is composed of a group of homogeneous cores (in this study) with a reconfigurable fabric. The advantage of DMPs over the traditional CMP is the ability to reconfigure the processor to better match the tasks at hand. For example, large sequential sections of code with high Instruction Level Parallelism (ILP) can be accelerated on a set of fused cores that mimic a wide superscalar processor. On a parallel workload the DMP can be reconfigured to match the Thread Level Parallelism (TLP).

In this paper we consider a dynamic multicore processor which allows cores to compose their execution resources, register files and private L1 caches to create logical processors to accelerate a single thread. Figure 1 shows a high-level view of the architecture and the two possible states: composed and decomposed. The composed state represents a set of physical cores fused to create a larger logical core. Multiple sets of cores can be fused to create logical cores of different sizes. In Figure 1 for example, LP1 is composed of four physical cores whereas LP2 is composed of two. At runtime, physical cores may be decomposed from a logical processor to remove them from the core composition.

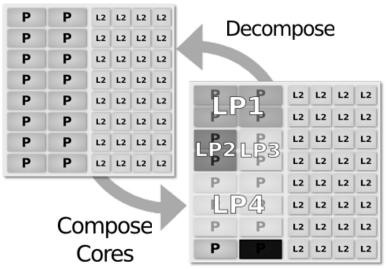

Figure 1: High-level view of a dynamic multicore processor considered in this paper.

2.2 Streaming Programming Languages

Streaming programming languages are a branch of dataflow programming that focus on applications that deal with a constant stream of data. These applications, such as audio or video decoding can be commonly found in mobile devices. Unlike conventional programming languages such as C++, these languages abstract the concept of incoming and outgoing data to permit the programmer to focus on how the data should be treated. Programs are described as directed graphs where nodes are functions and their edges represent their input and output streams. These languages offer primitives to describe such a graph [20] which expose parallelizable and serial sections of the application directly to the compiler. Rates of incoming and outcoming data can also be defined to facilitate load balancing optimizations [6].

Features of streaming programming languages make them an ideal language for targeting multicore processors. The explicit data communication between the different tasks in the program, the ability to estimate the amount of work performed in each task and information about data rates between tasks allows the compiler to easily generate a multi-threaded application that can run on a dynamic multicore processor. However, the main challenge consists of deciding how to map the different tasks onto threads and how to allocate the right amount of resources to maximize performance.

3. Motivation

This section illustrates the difficulty of finding a good partition and resource allocation. A simple experiment is conducted where we take one StreamIt benchmark, *Beamformer*, and partition its tasks into threads and allocate various number of cores to each thread. A co-design of more than 32,000 combinations (exhaustive space) of thread mappings and core compositions is generated. Each design point is executed on a dynamic multicore simulator (exact details about the experimental setup are presented later in section 4).

Figure 2 presents the distribution of the execution times from the co-design space as a violin plot. For the unfamiliar reader, an intuitive way to think about this violin plot is to consider it as a smoothed histogram rotated by 90 degrees and mirrored. We observed that the majority of the sampled points have a cycle count around 525,000 with the worst points taking more than 2 millions cycles. The best performance is around 275,000 cycles which is about 2x faster than the majority of the data points. This shows that finding the right combination of thread mapping and core composition is critical since a wrong choice often leads to suboptimal performance.

This example illustrates the necessity for designing the technique to predict the optimal number of threads and core composition to use. The next section will present a more in-depth analysis of the design space before presenting our machine-learning predictive model.

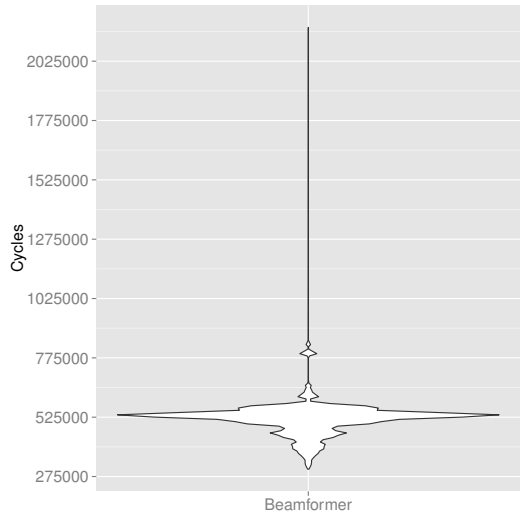

Figure 2: Distribution of the runtime for Beamformer resulting from an exhaustively exploration of the hardware/software co-design space. The application has been partitioned into different number of threads and core compositions.

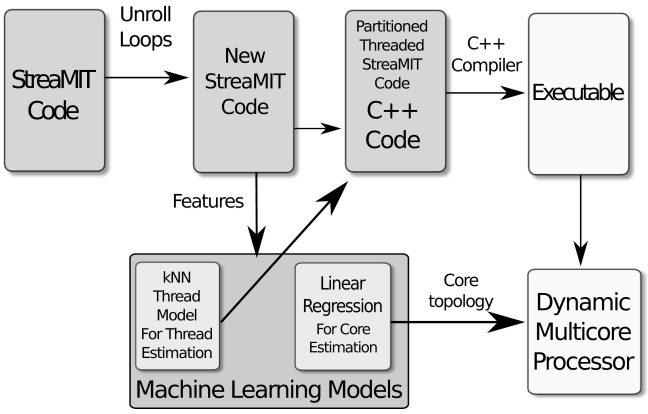

Figure 3: Description of our workflow. Two distinct machine-learning models are used to predict the optimal thread partitioning and core composition based on static code features.

4. Methodology

In this section we present our design exploration of a set of streaming applications being executed on a DMP. We describe how changing the thread mapping and core composition affect the benchmarks and what we can learn from this. In addition, we look at the impact of loop unrolling and how it helps exploit larger fused cores.

4.1 Overview

Figure 3 presents the workflow of our system. First, we use the source-to-source StreamIt compiler to unroll loops as this is usually beneficial when cores are composed as we will see later. Then, we extract static code features such as the program's graph structure. These features are used as an input to our first machine-learning model to determine the Thread Level Parallelism (TLP). This information is used to partition the program into threads and the StreamIt compiler produces a C++ program which is then compiled using our C++ compiler.

Parameter	Values
# of cores in the processor	16
# threads per application	1 – 15
# cores per thread	1 – 15
# sampled core compositions	100
# our sampled space	1316
# total sample space	32762

Table 1: Design space considered per application.

Then, a second machine-learning model is used which uses static code features extracted from the SteamIt code. This model is used to decide on the core topology. This is achieved by finding the amount of Instruction Level Parallelism (ILP) in each thread and by determining how many physical cores should be fused for that thread. Finally, we reconfigure the processor to fuse the requested resources and execute the partitioned program.

4.2 Dynamic Multicore Processor

We use a Dynamic Multicore Processor for our research based on an Explicit Data Graph Execution (EDGE) Instruction Set Architecture that resembles [10]. This differs from other DMPs such as CoreFusion, WidGET and Shared Architecture [12, 24, 26] which utilize a CISC/RISC instruction set. To evaluate our work we use a customizable cycle-level simulator verified within 4% of RTL. The simulator is highly configurable, allowing us to model a variety of parameters such as the number of cores, details of the memory hierarchy and synchronisation schemes. For our experiments we use a 16 core dual issue configuration with 16 KB private L1 caches and a 2 MB shared L2.

4.3 StreamIt Benchmarks

StreamIt is a high-level synchronous dataflow streaming programming language that defines programs as directed graphs. StreamIt offers an elegant way of describing streaming applications, abstracting away how infinite data streams are managed to allow the programmer to solely focus on how the data must be treated. A StreamIt program is composed of functions - called *Filters* - which operate on streams of data. Filters can be connected via *Pipelines*, *SplitJoins* or *Feedback Loops*.

Pipelines represent a sequence of connecting filters operating on the same stream, each filter operating on the output of the previous filter. In a SplitJoin, data in the stream is passed through a split filter and either duplicated and passed on in parallel to the filters or distributed amongst the filters in a round-robin manner. The output of all the filters in a SplitJoin are then concatenated in a round-robin fashion through a joiner filter. Finally a Feedback Loop provides a way for filters to operate on their outputs. The resulting program written in StreamIt represents a graph where the nodes are filters and their edges represent the incoming and outgoing data streams.

In this paper, we use 15 StreamIt benchmark all taken from the official StreamIt repository. For each benchmark we used the default input provided in the repository and the default iteration count of 10.

4.4 Design space

The parameters and size of the space are given in table 1. In this study we use 16 cores and assign core 0 to the main thread and for runtime management. This leaves 15 cores available for each application. We restrict each core to running only a single thread (no preemptive scheduling) which leads to a possible number of threads between 1 and 15. Cores can be fused together to form a logical core with up to 15 physical cores, making the total number

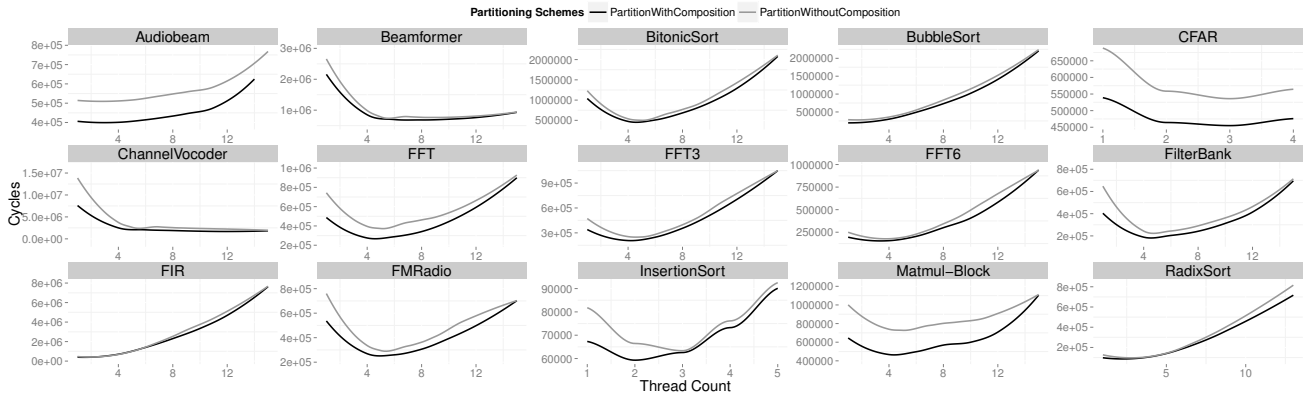

Figure 5: Performance as a function of the number of threads. The performance metric is number of cycles. Each benchmark has the performance measured with cores composed and with threads mapped to a single core.

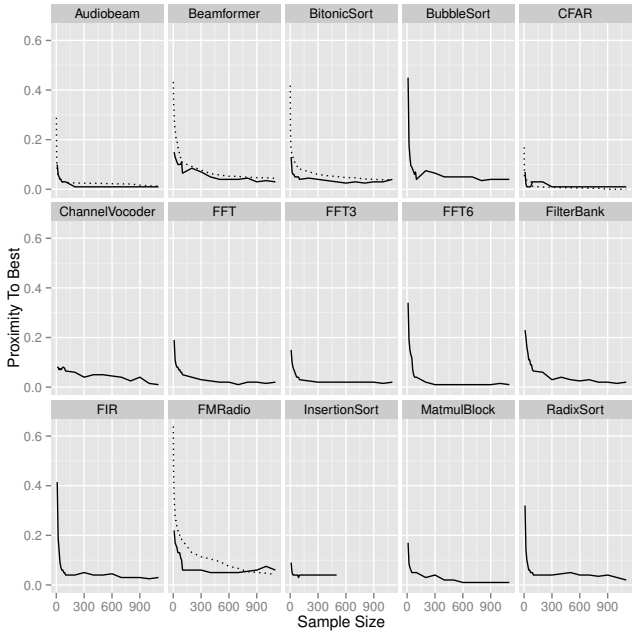

Figure 4: Statistical (plain line) and actual proximity (dotted line) of best performance in a subset of the 15 dimensional space to the best performance in the full space. This is a total space size of 32,767 unique combination per benchmark.

4.5 Sample Space

Given a partitioning, any benchmark that is split into 15 threads requires 32,767 executions to cover the entire space. Running an exhaustive exploration of the space requires approximately a week of simulation on a 572+ node supercomputer. For this reason, we decided to sample 1,316 random points from the entire space. This roughly corresponds to 100 core compositions for each number of threads (the actual number, 1,316 is smaller than 1,500 since for low thread counts there are less than 100 possible different core composition). *InsertionSort* is the only exception since it can at most only be split into 5 threads leading to 415 sample points.

To gain confidence that the best configuration from the sample space is indeed close to the real best in the entire space, we used a statistical model based on the Stopping Criterion defined in [21]. This model estimates, given a sample of the total space, if the best observed performance of that sample space is within a percentage of the statistical best performance. Our results demonstrate that the sample space selected is representative of the whole space.

Figure 4 shows, for each of the benchmarks, the proximity to the statistical best when increasing the sub-sample space given a maximal uncertainty of 5% (i. e. minimum 95% confidence). As can be seen by the plain line, the model shows that the best sample point is actually within 5% (0.05 proximity) of the best for all benchmark. To further prove that the statistical model based on the Stopping Criterion is indeed accurate, we conducted an exhaustive exploration for five benchmarks. The dotted line in figure 4 shows the actual proximity to the best for *Audiobeam*, *Beamformer*, *BitonicSort*, *CFAR* and *FMRadio*. As can be seen after 1316 samples, the performance we achieve is actually very similar to the one predicted by the statistical model, hence confirming prior work [21]. To summarize, we can conclude that the best point found in our sample space of 1,316 points is at least within 5% of the real best in the exhaustive space with 95% confidence.

5. Design Space Exploration

We now conduct an exploration of the software/hardware co-design space. The software side includes partitioning the program, determining the number of threads and the loop unrolling compiler optimization. The hardware side is about finding out the best core composition that maximizes performance for a given partitioning.

5.1 Thread Partitioning

We start by analyzing the impact of thread partitioning on performance. Thread partitioning is about deciding how many threads to create and how to partition StreamIt filters into these threads. To simplify this study, we use the default streaming partitioner to decide on how to allocate filters to cores which is based on simulated annealing. On the hardware side, we consider two scenarios: the "without composition scenario" where there is exactly one core per thread and the "with composition scenario" where each thread receives between 1 and 15 cores.

Figure 5 shows how performance varies under both scenarios as a function of the number of threads. We observe that regardless of how cores are composed all curves follow the same trend. The optimal number of threads using core composition is very similar to the scenario without composition. This important observation means that we can estimate the optimal number of threads for a bench-

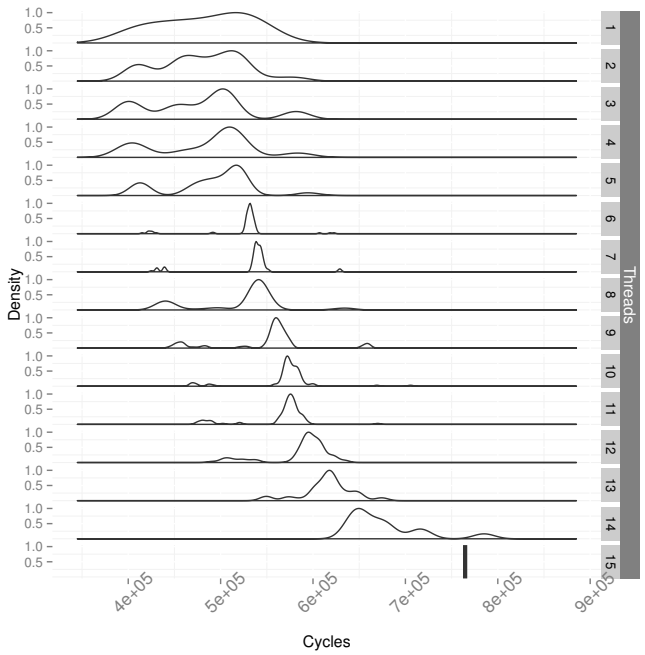

Figure 6: Distribution of Audiobeam performance when modifying the amount of threads and compositions.

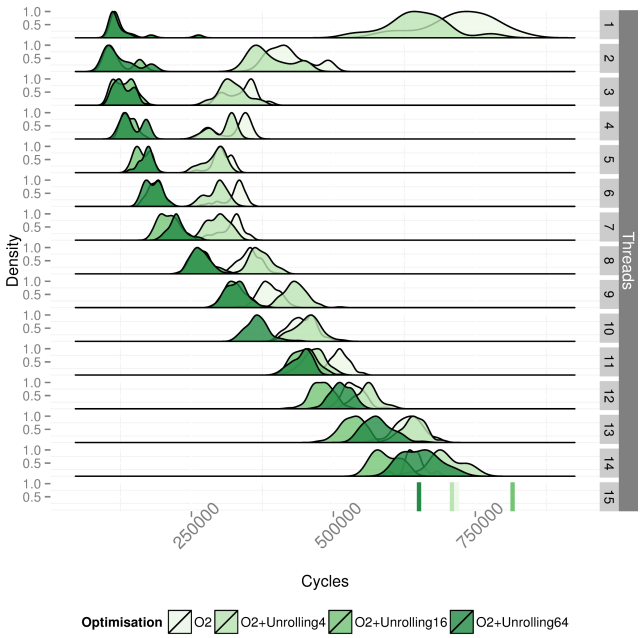

Figure 7: Distribution of FMRadio performance when modifying the amount of threads, composition and unrolling factor.

mark independently of the hardware composition. Our system can therefore proceed in two stages: first determine the optimal number of threads and then decide on a core composition.

Figure 5 also shows that the performance of most benchmarks starts deteriorating passed a certain number of threads making it critical to not over-allocate threads. This motivates our use of machine learning to decide the optimal number of threads to use. Finally we also observe that executions without compositions always perform worse. This demonstrates that composing cores is essential to obtain the best performance from a workload.

5.2 Core Composition

Using core composition, the processor fuses a number of cores and associates them to a thread to increase single threaded performance. Whilst this flexibility is advantageous, choosing the right amount of cores for a given thread is difficult due to the large number of possible configurations [11].

Figure 6 shows how threading and composition affects performance for the *Audiobeam* benchmark. The curves represent the density distribution for different core compositions as a function of the number of threads. The right hand side Y-axis represents the number of threads present in the current version of the benchmark normalized by the total number of points in the design space. For each of the threaded versions we ran the benchmark using on average 100 different compositions. The density curve for thread 15 is composed of a single point as there exists only a single composition for it.

The variance of each of the curves represents the influence of composition on the benchmark's performance for a given number of threads. For this benchmark the impact of core composition is actually very large for the best performing number of threads (1–5). Interestingly, as more threads are used, performance shifts worsens, echoing the results shown in the previous section.

5.3 Impact of Loop Unrolling

In this section we study the impact of one compiler optimization by focusing on loop unrolling. Filters containing large amounts of loops potentially contain high degrees of instruction level and memory level parallelism. Unrolling may increase the degree of parallelism which is advantageous to a wider fused processor. Loop unrolling may also yield similar results to vectorization when vectorization may not easily be applied or available.

Figure 7 presents an example of how loop unrolling affects performance on the *FMRadio* benchmark. The graph presents the same information as Figure 6 but with different executions of the benchmark when optimizing for speed and unroll factors 4, 16, and 64. Figure 7 shows that unrolling loops for *FMRadio* can greatly improve performance.

Another observation is that the best execution times for each of the threaded versions when unrolling does not follow the same trend previously described. The leftmost curve performance peaks at two threads whereas the rightmost peaks at five. As the number of cores fused can now be greater we encounter a resource problem when increasing the number of threads.

This example demonstrates that whilst the optimal number of threads is independent of the number of cores there still exists trade-offs between the two. This signifies that the amount of resources available to each thread must be taken into consideration before generating the program to balance the trade off between ILP and TLP.

5.4 Co-Design Space Best Results

This section presents the results of the entire co-design space exploration. Figure 8 characterizes how much of a performance increase is obtainable using a baseline of executing the benchmark on a single thread and single core without unrolling. For each benchmark, the *THREAD* bar represents the maximal speedup obtained by dividing the program into threads without fusing cores. The *CORE* bar represents the best speedup when we execute the benchmark in

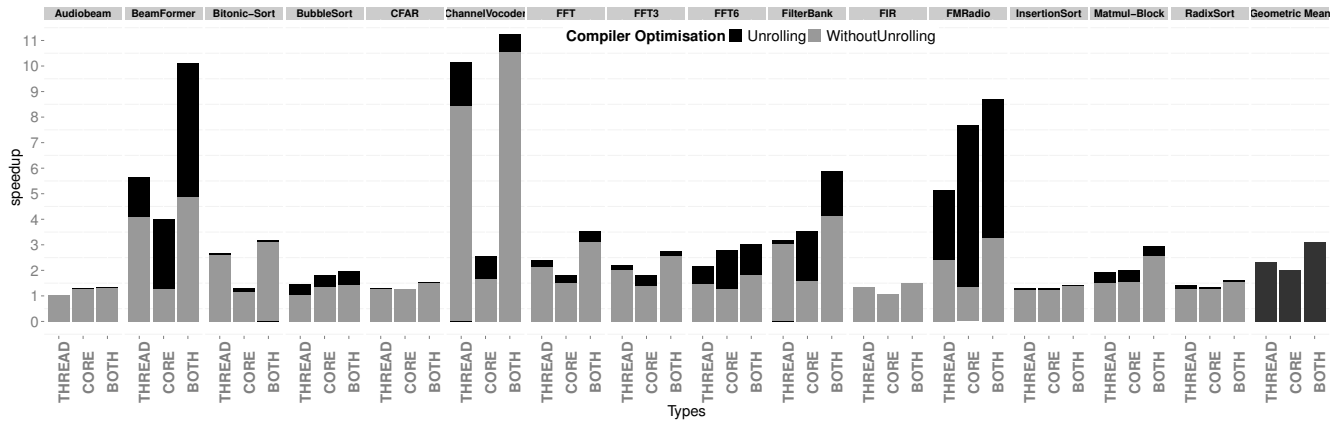

Figure 8: Speedup obtained by choosing best core composition, best thread number and the combination of both optimisations. The baseline for the speedup measurement is single core, single thread execution using O2 compiler optimisations. Higher is better.

a single thread and fuse cores. *BOTH* represents the best speedup obtained for each benchmark using a combination of *THREAD* and *CORE*. Finally, for each benchmark, we obtained these results for both an unrolled and not unrolled to compare how unrolling affects performance. Figure 8 shows that when loops are not unrolled, composing cores will not greatly improve performance.

When studying the geometric mean we see that, without unrolling, finding the correct number of threads gives a speedup of 1.92 compared to 1.33 when using only core composition. This changes when taking unrolling into account as the core compositions can be used more efficiently. In this case, the speedup obtained from only composing cores is 13% worse than using only threads. The unrolling demonstrates that the StreamIt programs must be modified to take advantage of the core composition. Finally, it is important to note that whilst finding the optimal thread mapping is better than the best composition, the best performance is always obtained through a combination of both optimizations.

5.5 Summary

This section demonstrated that each parameter has a large effect on the performance of the workload. We have seen that regardless of using core composition or not, there exists for each benchmark an optimal number of threads. Unrolling is effective at exposing more opportunities for composition due to increased ILP but there is a balance to strike between extracting ILP and TLP. Figure 8 shows there is a 3x benefit (overall) by automating the partitioning of both the software (threads) and hardware (cores).

6. Machine Learning Models

As seen in the previous section, selecting the right number of threads and a good combination of cores is difficult. This difficulty arises from trying to balance between exploiting larger composed cores with block speculation and ILP and between exploiting a larger number of logical cores via TLP.

The problem can be decomposed into two stages; first, determining the right number of threads and then selecting a good core composition. In this section, we present two machine-learning models that predict the best thread partitioning and core composition to maximize performance.

6.1 Predicting the Best Number of Threads

Synthetic Benchmark Generation One of the difficulties of building a machine learning based model for StreamIt is the lack of benchmarks available [23]. Whilst there exists at least 30 realistic applications for StreamIt [19] this is simply not enough to create a large enough data set. To overcome this problem we generate synthetic StreamIt benchmarks and gather statistics from them in a similar style as in [23]. To ensure that the synthetic benchmarks are representative of realistic benchmarks we created them using filters from a set of micro-kernels found in some StreamIt examples. We have 30 different possible filters with different incoming and outgoing rates, different inputs and outputs. We also ensured that the total number of filters and split joins found in a synthetic benchmark are within the average of the realistic benchmarks.

For each generated application, 15 different threaded versions are generated. Each of these versions is ran using a single core per thread and the cycle count is recorded. We repeated this for 1000 unique randomly generated applications and record the best number of threads each time.

Extracting Features Once the benchmarks have been generated, the next step consists of gathering features for each applications. In order to build our two machine learning models we used an initial set of over 50 features extracted from StreamIt programs. These features were extracted using pre-existing tools within StreamIt and some extra counters added by us. The features selected for our models were determine through correlation analysis. In this section, when discussing correlation we specifically look at which variables correlate with the optimal number of threads. These features are used by the model to make a prediction about the number of threads to use.

Figure 9 shows the 10 variables that correlate the most with the optimal thread number. In StreamIt the term multiplicity references the number of times a filter will have to execute in a time slice when the graph is in a steady state [9]. In Figure 9 the highest correlating value, Number of Distinct Multiplicities, determines all different multiplicities found in the StreamIt graph. Unconditionally executed blocks represent sets of operations in a filter that will always execute.

There are very little variables that highly correlate beyond Number of Distinct Multiplicities. A high number of distinct multiplicities implies that subsets of filters will execute at different rates. This means that certain filters may be local bottlenecks in a Pipeline for example. We suspect that when the number of distinct multiplicities is high this requires more threads to group filters with similar multiplicities. We can also see that the number of threads will depend on certain structural features such as Pipelines, SplitJoins and number of Filters. Yet, these variables seem to hold less influence on the number of threads a program needs than the different mul-

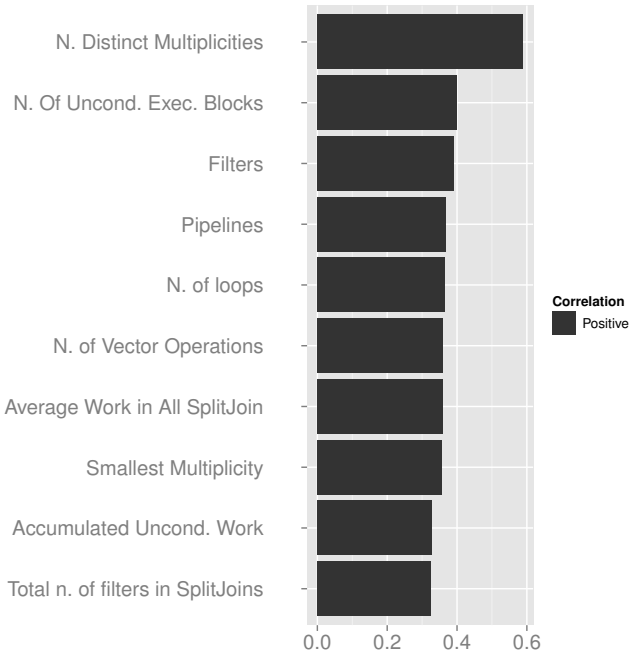

Figure 9: The ten highest correlating features with the best number of threads for 1000 synthetic benchmarks.

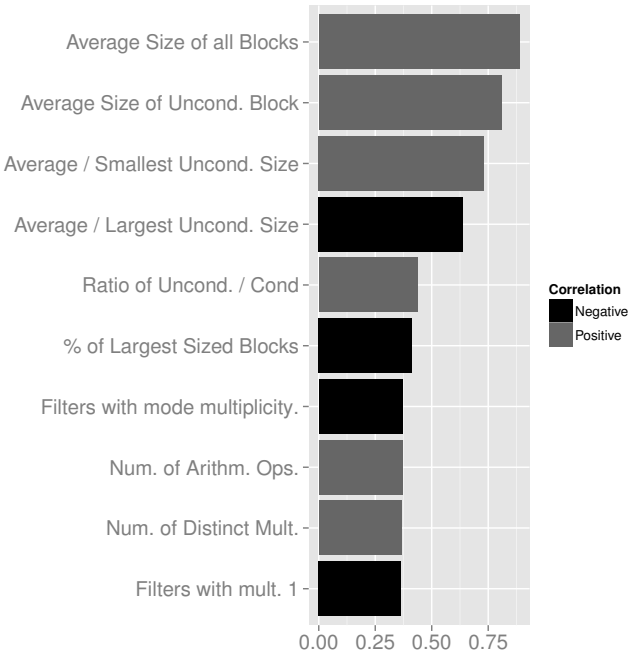

Figure 11: The ten highest correlating features with the optimal number of cores.

tiplicities found in the graph. This is most certainly due to the fact that whilst SplitJoins make parallelizable areas more visible, the amount of work contained in each stream of the SplitJoin, especially when this size is small, may actually make parallelizing the program worse due to ratio of communication to computation.

KNN Model We chose to use a k-Nearest Neighbor (kNN) to determine the number of threads to use for the application. Given a new application to predict, the kNN classifier determines the k closest generated applications in terms of the features. The distance between the features is measured using the Euclidean for each application. Once the set of k nearest neighbors has been identified, the model simply averages the best number of threads for each of the k nearest neighbors to make a prediction. The parameter k was determined experimentally using only the generated benchmarks. A value of $k = 7$ was found to lead to the best performance.

The features chosen are the variables displayed in Figure 9. Using cross validation we determine the efficiency by observing how close a classification is to our measured best thread number. We have determined that our model, using cross validation has a 33% accuracy of getting the predicted best thread number. This increases to 57% when we allow a prediction to be 1 thread away from the best and 67% when 2 threads away. Whilst the performance of pinpoint accuracy is disappointing we do not incur more than a 12% performance penalty when choosing a thread number which is +/- 1 from the best and 19% when moving up to 2 threads away from the best. This average is measured by looking at the thread performances without composition.

6.2 Predicting Core Composition

Gathering Training Data Given that the optimal number of cores for a thread is independent of the number of threads found in the program, we only use the single threaded versions to determine the optimal number of cores. For example, all benchmarks will only have a single core per thread when the application is partitioned in 15 threads as this is the maximum amount of cores that may be given to each thread rather than it being the optimal solution. We include multiple versions of the benchmarks using different amounts of unrolling. To determine the optimal number of cores we only select training data that has a performance within 1% of the best.

Analyzing Features Figure 11 shows the highest correlating features with the optimal number of cores. The features are very different from the ones presented in Figure 9 and overall there are higher correlating features. The highest correlating value has a correlation factor of 0.88 which represents the number of operations found in a basic block of code. The second feature is similar but only takes into account blocks that will be executed unconditionally, we have chosen to exclude blocks found in loops for this metric as there is still some form of condition for those blocks to be executed. The next two feature compare the size of the average size of an unconditional block to the largest and smallest unconditional block. The fifth feature measures the ratio of the number of unconditional blocks to conditional.

Overall there are no features distinct to StreamIt, such as pipelines or splitjoins that correlate highly with the optimal number of cores. We can thus infer that the optimal number of cores is independent of the structure of a StreamIt program. Instead, it is more dependent on the amount of computation.

EDGE architecture's ability to fetch atomic instruction blocks and out-of-order execution encourages the focus on determining how much speculation is extracted from each filter. Unfortunately StreamIt programs do not tend to have a large quantity of conditional statements and when they do they tend to be quite small. This statement is reinforced by the correlation between the average number of conditional blocks with the optimal number of cores, which is only 0.2, compared to 0.809 for the average size of unconditional blocks. We thus do not focus on using any speculative features from the StreamIt graph.

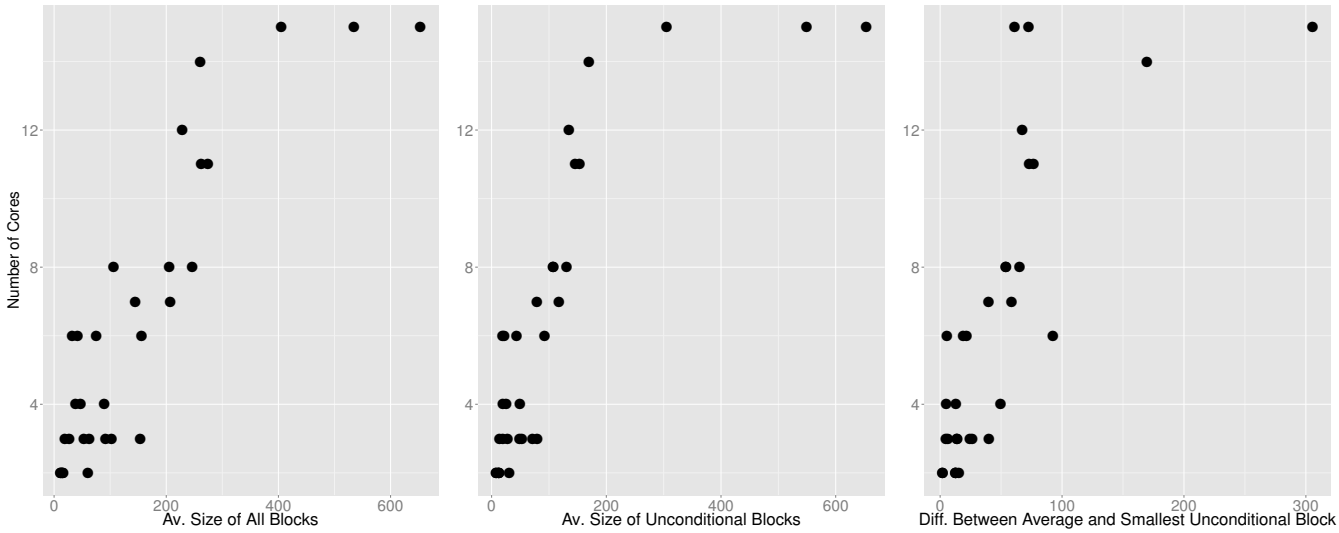

Figure 10: Optimal number of cores in relation to the three highest correlating features. The maximum number of cores plateaus on the right hand side as this is the maximum possible amount.

Linear Regression Model Given that the optimal number of cores is highly correlated with a few features, a linear regressor is a natural choice to predict the best number of threads. Figures 10 represent how the first three highest correlating values affect the number of cores. This figure was obtained by finding the best number of cores for a single threaded benchmark. It is important to note that the top right corner points will always be flat as we can only allocate a maximum of 15 cores.

7. Results

This section describes the performance achieved by the model when predicting the number of threads and core composition to use for each of the StreamIt benchmarks.

7.1 Evaluation Methodology

Leave-one-out cross-validation is used for testing the linear model. This means that when testing the model on one application, this application is removed from the training set, the model is trained with the remaining application and finally the model is tested on the application. This process is repeated for each application. This is standard methodology in the machine-learning community ensuring that the training data is never used for testing. For the kNN model, the training data consists of all the generated synthetic benchmarks and we only test it on the real StreamIt applications not used for training. To obtain the speedup we compare the performance of our machine learning based result and the best from the sample space to running the StreamIt benchmark on a single core, single thread, using O2 compiler optimisations.

7.2 Evaluation

Figure 12 compares the performance of the machine-learning model and the best performance from our sample space and core composition. As explained in the earlier section, the sampled best is drawn from a sample size of 1,316 combinations of core compositions and thread partitions for each application when possible. The baseline is the original StreamIt application running with one thread and one core on our dynamic multicore processor. The average speedup obtained through our machine learning model is 2.6, this is only 16% smaller than the average of the best found, which is a speedup of 3.1. These results are positive as it means we are at least within 16% of the total best. As can been seen in Figure 12 our largest performance penalty resides in the performance of *ChannelVocoder*.

Table 2 presents the actual configuration found for the best sampled point and the machine learning model prediction. Each column represent a different threads and the number in the cell represents the number of core associated with that thread. We can see that for *ChannelVocoder* our model predicts only 8 threads rather than the optimal 13. Refering back to Figure 5 and Figure 8 from Section 5 *ChannelVocoder* always performs better when adding threads. This is the cause of the performance penalty, for *ChannelVocoder* it is more important to allocate a higher number of threads rather than compose cores.

Aside from this case, our machine learning model obtains similar speedups to the best sample.

7.3 Summary

This section has demonstrated that it is possible to build a machine-learning model that achieves high level of performance using simple source code static features. In many applications, the model even comes very close to the best from the sampled space, showing that the features used by the model contain enough information to inform the model about the best decision.

8. Related Work

Dynamic Multicore Processors DMPs such as CoreFusion [12] differentiate themselves to EDGE based DMPs on their Instruction Set Architecture (ISA). CoreFusion uses a CISC/RISC based architecture which limits the degree of scalability (fusion), whereas EDGE based DMPs have shown promising scalability [10, 13]. Other types of DMPs such as WidGET [24] and Sharing Architecture [26] present a fine-grain level of composition. In these two architectures, cores can be created out of different components on the processor, including ALUs, floating point units and memory units. This differs from CoreFusion and EDGE where a logical core is composed out of a set of physical cores. This fine-grained composition can allow for even more optimisation but it increases the complexity of the problem.

	1	2	3	4	5	6	7	8	9	10
B Audiobeam	3	2								
M Audiobeam	2	3								
B Beamformer	1	4	2	4	4					
M Beamformer	6	4	4							
B BitonicSort	3	2	2	2						
M BitonicSort	1	2	2	1	2	2	2			
B BubbleSort	3	3								
M BubbleSort	2									
B CFAR	3	2								
M CFAR	2	2	1	2						
B ChannelVoc.	4	1	1	1	1	1	2	1	1	1
M ChannelVoc.	2	2	1	2	2	2	2			
B FIR	3	2								
M FIR	2	2								
B FFT	3	3	5							
M FFT	6	5	2							
B FFT3	3	2	2							
M FFT3	3	2	3	3	3	3				
B FFT6	7	8								
M FFT6	14									
B FilterBank	4	5	6							
M FilterBank	4	5								
B FMRadio	7	6								
M FMRadio	7	4								
B InsertionSort	3	2								
M InsertionSort	3									
B MatmulBlock	3	4	6	2						
M MatmulBlock	4	4								
B RadixSort	3	3								
M RadixSort	2	2								

Table 2: Number of Threads and Cores used for Best of Sample Space and Machine Learning Model.

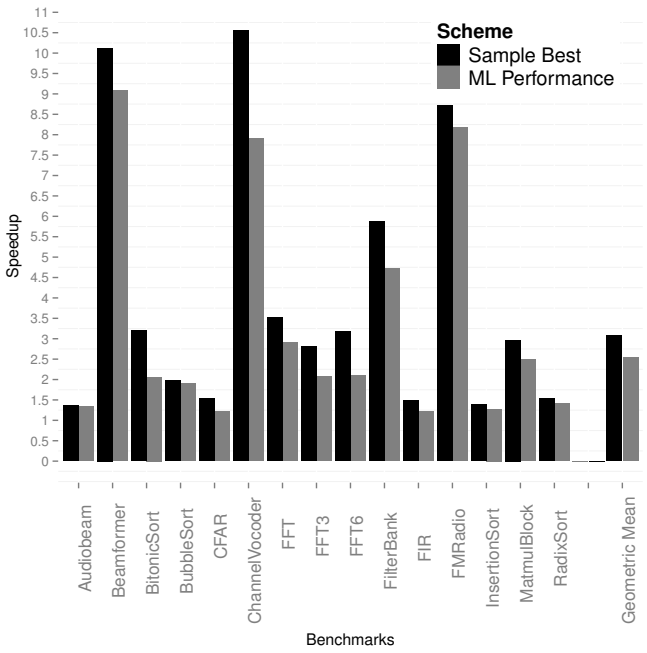

Figure 12: Performance of our machine learning model against the best execution from random sampling. The baseline for the speedup measurement is single core, single thread execution using O2 compiler optimisations. Higher is better.

Core Configuration Little work has been done on automatically determining the correct core composition for a given application. The work conducted in [12, 13] manually configure their processors before running benchmarks. In [17] they use information provided by the application to determine how to reconfigure some components of the processor. This initial information then assists the rest of the reconfiguration, this process still requires input from the programmer though. Therefore we present a novel method for automating the choice of core composition.

Streaming Programming Languages There exist streaming languages that target different architectures. For example Brook [4] is designed to be used on GPUs and WaveScript for embedded systems [15]. These languages present different constructs to StreamIt, in particular they lack the graph oriented constructs. Lacking such constructs make these languages less attractive for tile based processors.

Partitioning StreamIt on multicore chip Previous work on scheduling streaming applications onto DMPs or heterogenous multicore chips focuses on finding mathematical ways of partitioning the graph onto the chip [5, 14]. In Carpenter et al.'s work [5] they restrain themselves to partitioning a StreamIt application maintaining connectedness. Connectedness can be defined as a subgraph where the filters are connected. This restriction reduces the number of potential partitions that can be generated by their algorithm and will put TLP in favour of ILP. Kudlur et al. in [14] choose to represent the partitioning problem as an integer linear programming problem. They start by fissionioning stateless filters to obtain the optimal load balance across all cores and assign the filters to a core using a modulo scheduler. Farhad et al. also use integer linear programming in [8] to schedule StreamIt programs on multicore. They profile the communication costs of the streaming programs by running the program using different multicore allocations and feed that information into their integer linear programming model.

Machine Learning Using a machine learning model to partition StreamIt programs was previously explored in the work of Wang et al. in [23]. They use a k nearest neighbor model to determine the perfect partitioning of a StreamIt program for a multicore system. The features we extracted using correlation analysis are similar to those presented in the work of [23]. Unlike our work their model is used to find ways of fusing and fissioning filters to discover a new graph that can then be mapped onto a multicore system.

9. Conclusion

In this paper we presented the problem of partitioning both software and hardware for a Dynamic Multicore Processor. We analysed a set of streaming workloads based on StreamIt, extracting features which highly influence both the required number of threads and core composition. Using this data we introduced a machine learning model which is able to determine how many threads a StreamIt application needs and pick an appropriate chip topology. The model predicts configurations close to the performance of the best design points from the sampled space. By automating the decision of core composition we motivate the use of DMPs for accelerating applications without any involvement from the programmer.

Acknowledgements

This work has been supported by Microsoft Research through its PhD Scholarship Programme and has made use of the resources of the Edinburgh Compute and Data Facility (ECDF) [16].

References

[1] J. Auerbach, D. Bacon, I. Burcea, P. Cheng, S. Fink, R. Rabbah, and S. Shukla. A compiler and runtime for heterogeneous computing. In *DAC, 2012*, pages 271–276, June 2012.

[2] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *ISSCC 2008. IEEE International*, pages 88–598, Feb 2008.

[3] F. Bower, D. Sorin, and L. Cox. The impact of dynamically heterogeneous multicore processors on thread scheduling. *Micro, IEEE*, 28(3): 17–25, May 2008. ISSN 0272-1732. .

[4] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: Stream computing on graphics hardware. In *ACM SIGGRAPH 2004*, pages 777–786, New York, NY, USA, 2004. ACM.

[5] P. M. Carpenter, A. Ramirez, and E. Ayguade. Mapping stream programs onto heterogeneous multiprocessor systems. In *CASES '09*, pages 57–66, New York, NY, USA, 2009. ACM.

[6] J. Chen, M. I. Gordon, W. Thies, M. Zwicker, K. Pulli, and F. Durand. A reconfigurable architecture for load-balanced rendering. In *HWWS '05*, pages 71–80, New York, NY, USA, 2005. ACM.

[7] S. Eyerman and L. Eeckhout. Modeling critical sections in amdahl's law and its implications for multicore design. *SIGARCH Comput. Archit. News*, 38(3):362–370, June 2010. .

[8] S. M. Farhad, Y. Ko, B. Burgstaller, and B. Scholz. Profile-guided deployment of stream programs on multicores. LCTES '12, pages 79–88, New York, NY, USA, 2012. ACM. .

[9] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. *SIGARCH Comput. Archit. News*, 30(5):291–303, Oct. 2002. ISSN 0163-5964. .

[10] M. Govindan, B. Robatmili, D. Li, B. Maher, A. Smith, S. W. Keckler, and D. Burger. Scaling power and performance via processor composability. *IEEE Transactions on Computers*, 63(8):2025–2038, 2014.

[11] D. P. Gulati, C. Kim, S. Sethumadhavan, S. W. Keckler, and D. Burger. Multitasking workload scheduling on flexible core chip multiprocessors. *SIGARCH Comput. Archit. News*, 36(2):46–55, May 2008.

[12] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core fusion: Accommodating software diversity in chip multiprocessors. *SIGARCH Comput. Archit. News*, 35(2):186–197, June 2007.

[13] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable lightweight processors. In *MICRO '07*, pages 381–394, Washington, DC, USA, 2007. IEEE Computer Society.

[14] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. *SIGPLAN Not.*, 43(6):114–124, June 2008.

[15] R. R. Newton, L. D. Girod, M. B. Craig, S. R. Madden, and J. G. Morrisett. Design and evaluation of a compiler for embedded stream programs. In *LCTES '08*, pages 131–140, New York, NY, USA, 2008. ACM.

[16] U. of Edinburgh. Edinburgh compute and data facility web site, 1 August 2007, accessed 4th of April. 2016. `www.ecdf.ed.ac.uk`.

[17] P. Santos, G. Nazar, F. Anjam, S. Wong, D. Matos, and L. Carro. A fully dynamic reconfigurable noc-based mpsoc: The advantages of total reconfiguration. In *HiPEAC '13*, Berlin, Germany, January 2013.

[18] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. *SIGPLAN Not.*, 44(3):253–264, Mar. 2009.

[19] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *PACT '10*, pages 365–376, New York, NY, USA, 2010. ACM.

[20] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *CC*, pages 179–196, London, UK, UK, 2002. Springer-Verlag.

[21] R. W. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, 2003. AAI3121741.

[22] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *Computer*, 30 (9):86–93, Sep 1997.

[23] Z. Wang and M. F. P. O'boyle. Using machine learning to partition streaming programs. *ACM Trans. Archit. Code Optim.*, 10(3):20:1–20:25, Sept. 2008.

[24] Y. Watanabe, J. D. Davis, and D. A. Wood. Widget: Wisconsin decoupled grid execution tiles. *SIGARCH Comput. Archit. News*, 38 (3):2–13, June 2010.

[25] P. M. Wells, K. Chakraborty, and G. S. Sohi. Dynamic heterogeneity and the need for multicore virtualization. *SIGOPS Oper. Syst. Rev.*, 43 (2):5–14, Apr. 2009.

[26] Y. Zhou and D. Wentzlaff. The sharing architecture: Sub-core configurability for iaas clouds. *SIGPLAN Not.*, 49(4):559–574, Feb. 2014.